05 | 调度系统：“数据不动代码动”到底是什么意思？

2021-03-24 吴磊

Spark性能调优实战 进入课程 >

讲述：吴磊
时长 25:07 大小 23.00M

你好，我是吴磊。

在日常的开发与调优工作中，为了充分利用硬件资源，我们往往需要手工调节任务并行度来提升 CPU 利用率，控制任务并行度的参数是 Spark 的配置项：spark.default.parallelism。增加并行度确实能够充分利用闲置的 CPU 线程，但是，parallelism 数值也不宜过大，过大反而会引入过多的调度开销，得不偿失。

这个调优技巧可以说是老生常谈了，网上到处都可以搜得到。那你知道为什么 parallelism 数值过大调度开销会呈指数级增长吗？调度开销具体又是指什么呢？以及，如果不想一个数值一个数值的尝试，parallelism 数值究竟该怎么设置，才能以最少的时间获得最好的效果？如果你还没有答案，或者说还没有把握答对，接下来你就要好好听我讲。

这一讲，我会通过一个机器学习案例，来和你一起聊聊调度系统是什么，它是怎么工作的，从而帮助你摆脱调优总是停留在知其然、不知其所以然的尴尬境地。

案例：对用户兴趣特征做 Label Encoding

在机器学习应用中，特征工程几乎占据了算法同学 80% 的时间和精力，毕竟，一份质量优良的训练样本限定了模型效果的上限和天花板，我们要讲的案例就来自特征工程中一个典型的处理场景：Label Encoding（标签编码）。

什么是 Label encoding 呢？模型特征按照是否连续可以分为两类：连续性数值特征和离散型特征，离散型特征往往以字符串的形式存在，比如用户兴趣特征就包括体育、政治、军事和娱乐等。对于很多机器学习算法来说，字符串类型的数据是不能直接消费的，需要转换为数值才行，例如把体育、政治、军事、娱乐映射为 0、1、2、3，这个过程在机器学习领域有个术语就叫 Label encoding。

我们这一讲的案例，就是要对用户兴趣特征做 Label encoding，简单来说就是以固定的模板把字符串转换为数值，然后将千亿条样本中的用户兴趣转换为对应的索引值。固定模板是离线模型训练与线上模型服务之间的文件接口，内容仅包含用户兴趣这一列，字符串已按事先约定好的规则进行排序。我们需要注意的是，用户兴趣包含 4 个层级，因此这个模板文件较大，记录数达到万级别。

复制代码

```
//模板文件
//用户兴趣
体育-篮球-NBA-湖人
军事-武器-步枪-AK47
```

那具体怎么转换呢？例如，我们可以将用户兴趣“体育 - 篮球 -NBA- 湖人”映射为 0，将兴趣“军事 - 武器 - 步枪 -AK47”映射为 1，以此类推。应该说，需求还是相当明确的，我身边的同学们拿到需求之后，奔儿都没打，以迅雷不及掩耳之势就实现了如下的处理函数。

复制代码

```
/**
实现方式1
输入参数：模板文件路径，用户兴趣字符串
返回值：用户兴趣字符串对应的索引值
```

```
*/

//函数定义
def findIndex(templatePath: String, interest: String): Int = {
val source = Source.fromFile(filePath, "UTF-8")
val lines = source.getLines().toArray
source.close()
val searchMap = lines.zip(0 until lines.size).toMap
searchMap.getOrElse(interest, -1)
}

//Dataset中的函数调用
findIndex(filePath, "体育-篮球-NBA-湖人")
```

我们可以看到这个函数有两个形参，一个是模板文件路径，另一个是训练样本中的用户兴趣。处理函数首先读取模板文件，然后根据文件中排序的字符串构建一个从兴趣到索引的 Map 映射，最后在这个 Map 中查找第二个形参传入的用户兴趣，如果能找到则返回对应的索引，找不到的话则返回 -1。

这段代码看上去似乎没什么问题，同学们基于上面的函数对千亿样本做 Label encoding，在 20 台机型为 C5.4xlarge AWS EC2 的分布式集群中花费了 5 个小时。坦白说，这样的执行性能，我是不能接受的。你可能会说：“需求就是这个样子，还能有什么别的办法呢？”我们不妨来看看另外一种实现方式。

复制代码

```
/**
实现方式2
输入参数：模板文件路径，用户兴趣字符串
返回值：用户兴趣字符串对应的索引值
*/

//函数定义
val findIndex: (String) => (String) => Int = {
(filePath) =>
val source = Source.fromFile(filePath, "UTF-8")
val lines = source.getLines().toArray
source.close()
val searchMap = lines.zip(0 until lines.size).toMap
(interest) => searchMap.getOrElse(interest, -1)
}
val partFunc = findIndex(filePath)

//Dataset中的函数调用
partFunc("体育-篮球-NBA-湖人")
```

同学们基于第二种方式对相同的数据集做 Label encoding 之后，在 10 台同样机型的分布式集群中花了不到 20 分钟就把任务跑完了。可以说，执行性能的提升是显而易见的。那么，两份代码有什么区别呢？

我们可以看到，相比于第一份代码，第二份代码的函数体内没有任何变化，还是先读取模板文件、构建 Map 映射、查找用户兴趣，最后返回索引。最大的区别就是第二份代码对高阶函数的使用，具体来说有 2 点：

1. 处理函数定义为高阶函数，形参是模板文件路径，返回结果是从用户兴趣到索引的函数；
2. 封装千亿样本的 Dataset 所调用的函数，不是第一份代码中的 findIndex，而是用模板文件调用 findIndex 得到的 partFunc，partFunc 是形参为兴趣、结果为索引的普通标量函数。

那么，高阶函数真有这么神奇吗？其实，性能的提升并不是高阶函数的功劳，而是调度系统在起作用。

Spark 的调度系统是如何工作的？

Spark 调度系统的核心职责是，**先将用户构建的 DAG 转化为分布式任务，结合分布式集群资源的可用性，基于调度规则依序把分布式任务分发到执行器**。这个过程听上去就够复杂的了，为了方便你理解，我们还是先来讲一个小故事。

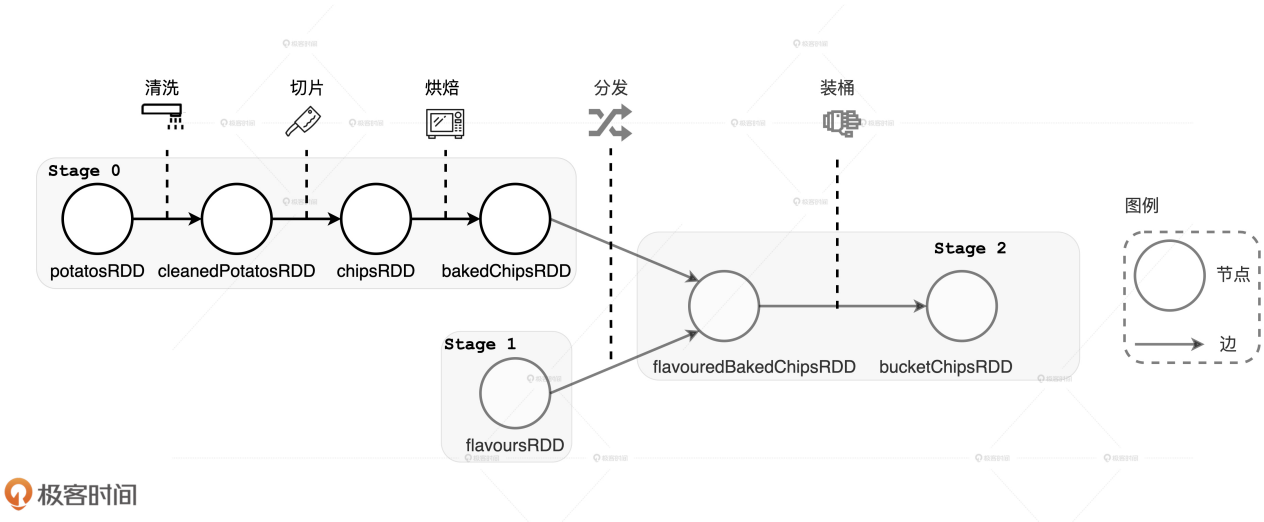
土豆工坊流水线升级

在学完了内存计算的第二层含义之后，土豆工坊的老板决定对土豆加工流水线做升级，来提高工坊的生产效率和灵活性。

这里，我们先对内存计算的第二层含义做个简单地回顾，它指的是**同一 Stage 中的所有操作会被捏合为一个函数，这个函数一次性会被地应用到输入数据上，并且一次性地产生计算结果**。

升级之前的土豆加工流程 DAG 被切分为 3 个执行阶段 Stage，它们分别是 Stage 0、Stage 1、Stage 2。其中，Stage 0 产出即食薯片，Stage 1 分发调味品，Stage 2 则产

出不同尺寸、不同风味的薯片。我们重点关注 Stage 0，Stage 0 有 3 个加工环节，分别是清洗、切片和烘焙。这 3 个环节需要 3 种不同的设备，即清洗机、切片机和烤箱。



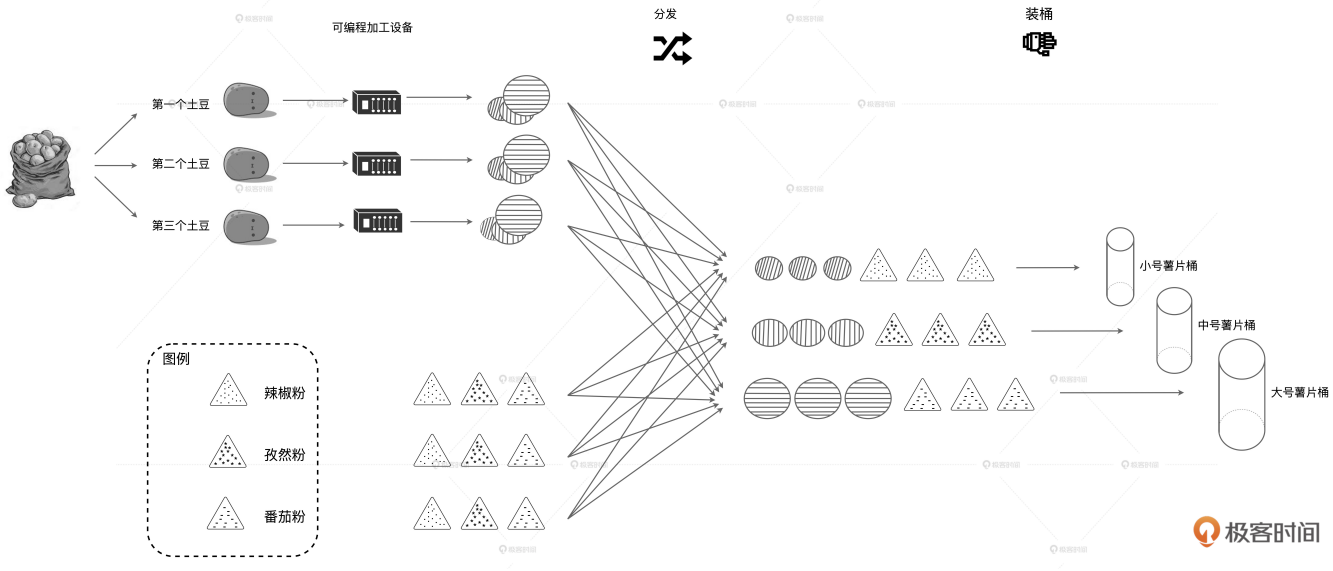
土豆工坊加工流程的3个执行阶段

工坊有 3 条流水线，每种设备都需要 3 套，在成本方面要花不少钱呢，因此工坊老板一直绞尽脑汁想把设备方面的成本降下来。

此时，工头儿建议：“老板，我听说市场上有一种可编程的土豆加工设备，它是个黑盒子并且只有输入口和输出口，从外面看不见里面的操作流程。不过黑盒子受程序控制，给定输入口的食材，我们可以编写程序控制黑盒子的输出。有了这个可编程设备，咱们不但省了钱，将来还可以灵活地扩充产品线。比方想生产各种风味的薯条或是土豆泥，只需要更换一份程序加载到黑盒子里就行啦！”

老板听后大喜，决定花钱购入可编程土豆加工设备，替换并淘汰现有的清洗机、切片机和烤箱。

于是，工坊的加工流水线就变成了如下的样子。工人们的工作也从按照 DAG 流程图的关键步骤，在流水线上安装相应的设备，变成了把关键步骤编写相应的程序加载到黑盒内。这样一来，这家工坊的生产力也从作坊式的生产方式，升级到了现代化流水线的作业模式。



演进的土豆加工流水线

那么，这个故事跟我们今天要讲的调度系统有什么关系呢？事实上，Spark 调度系统的工作流程包含如下 5 个步骤：

1. 将 DAG 拆分为不同的运行阶段 Stages；

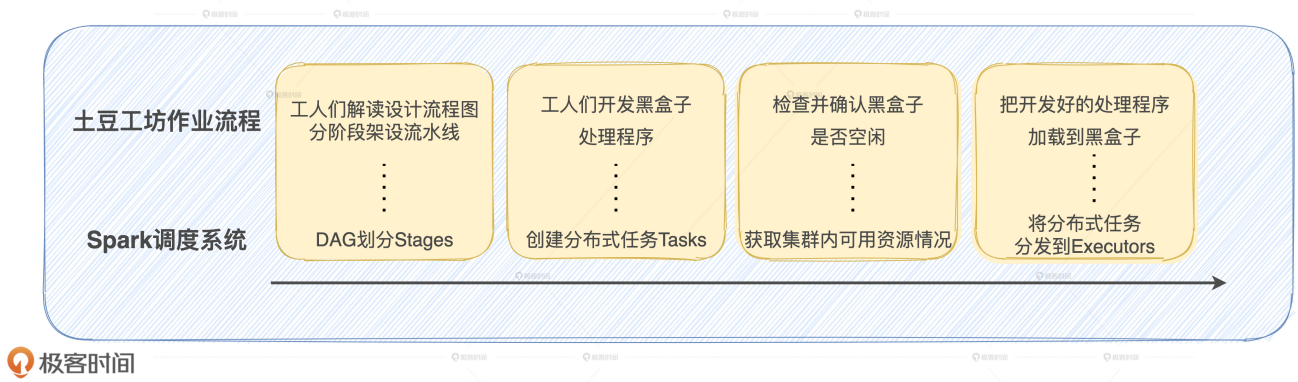
2. 创建分布式任务 Tasks 和任务组 TaskSet；

3. 获取集群内可用的硬件资源情况；

4. 按照调度规则决定优先调度哪些任务 / 组；

5. 依序将分布式任务分发到执行器 Executor。

除了第 4 步以外，其他几步和土豆工坊流水线上的关键步骤都是一一对应的，它们的对应关系如下：



现在，你可能会觉得用故事来记这几个步骤好像多此一举，但当我们学完了所有的原理之后，再回过头来把故事的主线串联起来，你就会惊喜地发现，所有的原理你都能轻松地记住和理解，这可比死记硬背的效率要高得多。

调度系统中的核心组件有哪些?

接下来，我们深入到流程中的每一步去探究 Spark 调度系统是如何工作的。不过在此之前，我们得先弄清楚调度系统都包含哪些关键组件，不同组件之间如何交互，它们分别担任了什么角色，才能更好地理解流程中的每一步。

Spark 调度系统包含 3 个核心组件，分别是 DAGScheduler、TaskScheduler 和 SchedulerBackend。这 3 个组件都运行在 Driver 进程中，它们通力合作将用户构建的 DAG 转化为分布式任务，再把这些任务分发给集群中的 Executors 去执行。不过，它们的名字都包含 Scheduler，光看名字还真是丈二和尚摸不着头脑，所以我把它们和调度系统流程中 5 个步骤的对应关系总结在了下表中，你可以看一看。

调度系统流程步骤	调度系统核心组件
将DAG拆分为不同的运行阶段Stages	DAGScheduler
创建分布式任务Tasks和任务组TaskSet	DAGScheduler
获取集群内可用硬件资源情况	SchedulerBackend
按照调度规则决定优先调度哪些任务/组	TaskScheduler
依序将分布式任务分发到执行器Executor	SchedulerBackend

极客时间

1. DAGScheduler

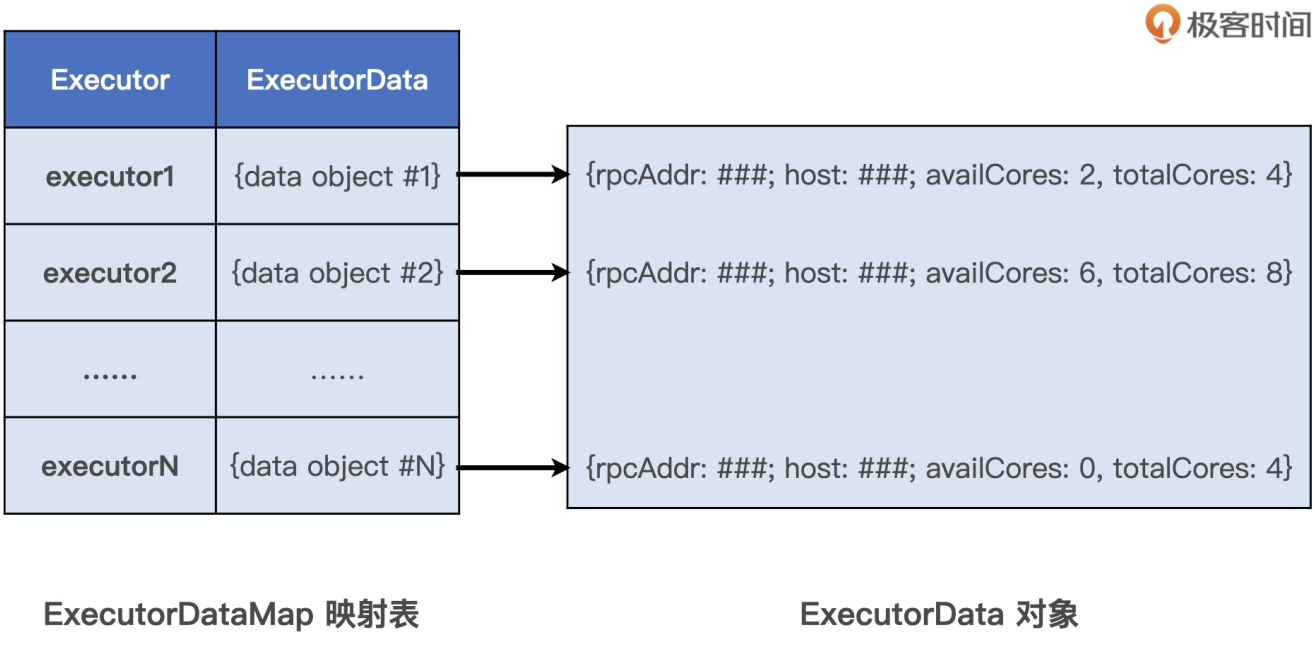
DAGScheduler 的主要职责有二：一是把用户 DAG 拆分为 Stages，如果你不记得这个过程可以回顾一下上一讲的内容；二是在 Stage 内创建计算任务 Tasks，这些任务囊括了用户通过组合不同算子实现的数据转换逻辑。然后，执行器 Executors 接收到 Tasks，会将其中封装的计算函数应用于分布式数据分片，去执行分布式的计算过程。

不过，如果我们给集群中处于繁忙或者是饱和状态的 Executors 分发了任务，执行效果会大打折扣。因此，**在分发任务之前，调度系统得先判断哪些节点的计算资源空闲，然后再把任务分发过去**。那么，调度系统是怎么判断节点是否空闲的呢？

2. SchedulerBackend

SchedulerBackend 就是用来干这个事的，它是对于资源调度器的封装与抽象，为了支持多样的资源调度模式如 Standalone、YARN 和 Mesos，SchedulerBackend 提供了对应的实现类。在运行时，Spark 根据用户提供的 MasterURL，来决定实例化哪种实现类的对象。MasterURL 就是你通过各种方式指定的资源管理器，如 --master spark://ip:host（Standalone 模式）、--master yarn（YARN 模式）。

对于集群中可用的计算资源，SchedulerBackend 会用一个叫做 ExecutorDataMap 的数据结构，来记录每一个计算节点中 Executors 的资源状态。ExecutorDataMap 是一种 HashMap，它的 Key 是标记 Executor 的字符串，Value 是一种叫做 ExecutorData 的数据结构，ExecutorData 用于封装 Executor 的资源状态，如 RPC 地址、主机地址、可用 CPU 核数和满配 CPU 核数等等，它相当于是对 Executor 做的“资源画像”。



ExecutorDataMap映射表

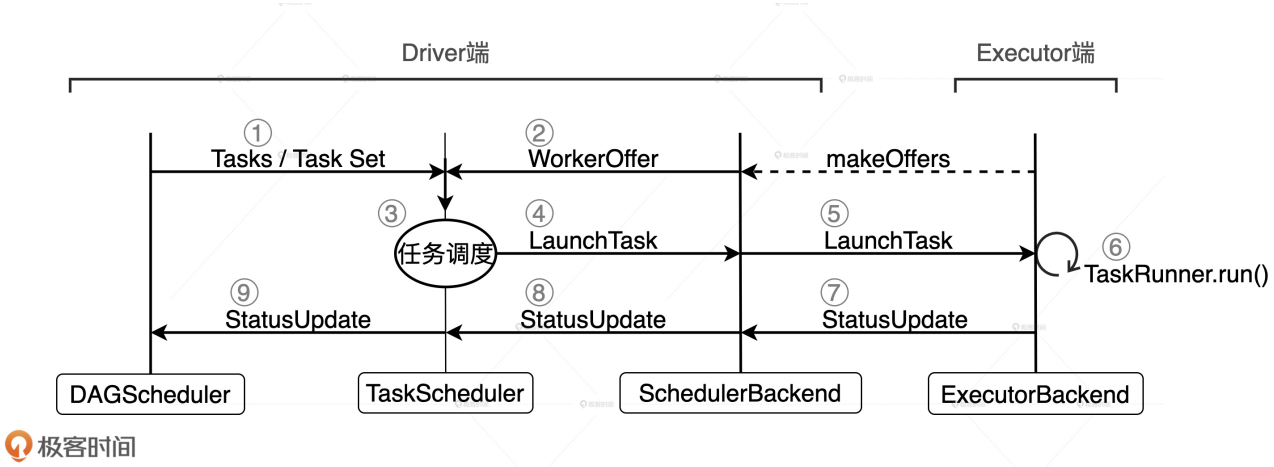
总的来说，对内，SchedulerBackend 用 ExecutorData 对 Executor 进行资源画像；对外，SchedulerBackend 以 WorkerOffer 为粒度提供计算资源，WorkerOffer 封装了 Executor ID、主机地址和 CPU 核数，用来表示一份可用于调度任务的空闲资源。显然，基于 Executor 资源画像，SchedulerBackend 可以同时提供多个 WorkerOffer 用于分布

式任务调度。WorkerOffer 这个名字起得蛮有意思，Offer 的字面意思是公司给你提供的工作机会，结合 Spark 调度系统的上下文，就变成了使用硬件资源的机会。

好了，到此为止，要调度的计算任务有了，就是 DAGScheduler 通过 Stages 创建的 Tasks；可用于调度任务的计算资源也有了，即 SchedulerBackend 提供的一个又一个 WorkerOffer。如果从供需的角度看待任务调度，DAGScheduler 就是需求端，SchedulerBackend 就是供给端。

3. TaskScheduler

左边有需求，右边有供给，如果把 Spark 调度系统看作是一个交易市场的话，那么中间还需要有个中介来帮它们对接意愿、撮合交易，从而最大限度地提升资源配置的效率。在 Spark 调度系统中，这个中介就是 TaskScheduler。**TaskScheduler 的职责是，基于既定的规则与策略达成供需双方的匹配与撮合。**



Spark分布式任务调度流程

显然，TaskScheduler 的核心是任务调度的规则和策略，**TaskScheduler 的调度策略分为两个层次，一个是不同 Stages 之间的调度优先级，一个是 Stages 内不同任务之间的调度优先级。**

首先，对于两个或多个 Stages，如果它们彼此之间不存在依赖关系、互相独立，在面对同一份可用计算资源的时候，它们之间就会存在竞争关系。这个时候，先调度谁、或者说谁优先享受这份计算资源，大家就得基于既定的规则和协议照章办事了。

对于这种 Stages 之间的任务调度，TaskScheduler 提供了 2 种调度模式，分别是 FIFO（先到先得）和 FAIR（公平调度）。 FIFO 非常好理解，在这种模式下，Stages 按

照被创建的时间顺序来依次消费可用计算资源。这就好比在二手房交易市场中，两个人同时看中一套房子，不管两个人各自愿意出多少钱，谁最先交定金，中介就优先给谁和卖家撮合交易。

你可能会说：“这不合常理啊！如果第二个人愿意出更多的钱，卖家自然更乐意和他成交。”没错，考虑到开发者的意愿度，TaskScheduler 提供了 FAIR 公平调度模式。在这种模式下，哪个 Stages 优先被调度，取决于用户在配置文件 fairscheduler.xml 中的定义。

在配置文件中，Spark 允许用户定义不同的调度池，每个调度池可以指定不同的调度优先级，用户在开发过程中可以关联不同作业与调度池的对应关系，这样不同 Stages 的调度就直接和开发者的意愿挂钩，也就能享受不同的优先级待遇。对应到二手房交易的例子中，如果第二个人乐意付 30% 的高溢价，中介自然乐意优先撮合他与卖家的交易。

说完了不同 Stages 之间的调度优先级，我们再来说说同一个 Stages 内部不同任务之间的调度优先级，Stages 内部的任务调度相对来说简单得多。**当 TaskScheduler 接收到来自 SchedulerBackend 的 WorkerOffer 后，TaskScheduler 会优先挑选那些满足本地性级别要求的任务进行分发。**众所周知，本地性级别有 4 种：Process local < Node local < Rack local < Any。从左到右分别是进程本地性、节点本地性、机架本地性和跨机架本地性。从左到右，计算任务访问所需数据的效率越来越差。

进程本地性表示计算任务所需的输入数据就在某一个 Executor 进程内，因此把这样的计算任务调度到目标进程内最划算。同理，如果数据源还未加载到 Executor 进程，而是存储在某一计算节点的磁盘中，那么把任务调度到目标节点上去，也是一个不错的选择。再次，如果我们无法确定输入源在哪台机器，但可以肯定它一定在某个机架上，本地性级别就会退化到 Rack local。

DAGScheduler 划分 Stages、创建分布式任务的过程中，会为每一个任务指定本地性级别，本地性级别中会记录该任务有意向的计算节点地址，甚至是 Executor 进程 ID。换句话说，**任务自带调度意愿，它通过本地性级别告诉 TaskScheduler 自己更乐意被调度到哪里去。**

既然计算任务的个人意愿这么强烈，TaskScheduler 作为中间商，肯定要优先满足人家的意愿。这就像一名码农想要租西二旗的房子，但是房产中介 App 推送的结果都是东三环国

贸的房子，那么这个中介的匹配算法肯定有问题。

由此可见，**Spark 调度系统的原则是尽可能地让数据呆在原地、保持不动，同时尽可能地把承载计算任务的代码分发到离数据最近的地方，从而最大限度地降低分布式系统中的网络开销**。毕竟，分发代码的开销要比分发数据的代价低太多，这也正是“数据不动代码动”这个说法的由来。

总的来说，TaskScheduler 根据本地性级别遴选出待计算任务之后，先对这些任务进行序列化。然后，交给 SchedulerBackend，SchedulerBackend 根据 ExecutorData 中记录的 RPC 地址和主机地址，再将序列化的任务通过网络分发到目的主机的 Executor 中去。最后，Executor 接收到任务之后，把任务交由内置的线程池，线程池中的多线程则并发地在不同数据分片之上执行任务中封装的数据处理函数，从而实现分布式计算。

性能调优案例回顾

知道了调度系统是如何工作的，我们就可以回过头来说说开头 Label encoding 的开发案例中，2 种实现方式的差别到底在哪儿了。我们先来回顾案例中处理函数的主要计算步骤：

1. 读取并遍历模板文件内容，建立从字符串到数值的字典；
2. 根据样本中的用户兴趣，查找字典并返回兴趣字符串对应的数值索引。

2 种实现方式的本质区别在于，函数中 2 个计算步骤的分布式计算过程不同。在第 1 种实现方式中，函数是一个接收两个形参的普通标量函数，Dataset 调用这个函数在千亿级样本上做 Label encoding。

在 Spark 任务调度流程中，该函数在 Driver 端交由 DAGScheduler 打包为 Tasks，经过 TaskScheduler 调度给 SchedulerBackend，最后由 SchedulerBackend 分发到集群中的 Executors 中去执行。这意味着集群中的每一个 Executors 都需要执行函数中封装的两个计算步骤，要知道，第一个步骤中遍历文件内容并建立字典的计算开销还是相当大的。

反观第 2 种实现方式，2 个计算步骤被封装到一个高阶函数中。用户代码先在 Driver 端用模板文件调用这个高阶函数，完成第一步计算建立字典的过程，同时输出一个只带一个形

参的标量函数，这个标量函数内携带了刚刚建好的映射字典。最后，Dataset 将这个标量函数作用于千亿样本之上做 Label encoding。

发现区别了吗？在第 2 种实现中，函数的第一步计算只在 Driver 端计算一次，分发给集群中所有 Executors 的任务中封装的是携带了字典的标量函数。然后在 Executors 端，Executors 在各自的数据分片上调用该函数，省去了扫描模板文件、建立字典的开销。最后，我们只需要把样本中的用户兴趣传递进去，函数就能以 O(1) 的查询效率返回数值结果。

对于一个有着成百上千 Executors 的分布式集群来说，这 2 种不同的实现方式带来的性能差异还是相当可观的。因此，如果你能把 Spark 调度系统的工作原理牢记于心，我相信在代码开发或是 review 的过程中，你都能够意识到第一个计算步骤会带来的性能问题。**这种开发过程中的反思，其实就是在潜移默化地建立以性能为导向的开发习惯。**

小结

今天这一讲，我们先通过一个机器学的案例对比了 2 种实现方式的性能差异，知道了对于调度系统一知半解，很有可能在开发过程中引入潜在的性能隐患。为此，我梳理了调度系统工作流程的 5 个主要步骤：

1. 将 DAG 拆分为不同的运行阶段 Stages；
2. 创建分布式任务 Tasks 和任务组 TaskSet；
3. 获取集群内可用硬件资源情况；
4. 按照调度规则决定优先调度哪些任务 / 组；
5. 依序将分布式任务分发到执行器 Executor；

结合这 5 个步骤，我们深入分析了 Spark 调度系统的工作原理，我们可以从核心职责和核心原则这两方面来归纳：

1. Spark 调度系统的核心职责是，先将用户构建的 DAG 转化为分布式任务，结合分布式集群资源的可用性，基于调度规则依序把分布式任务分发到执行器 Executors；
2. Spark 调度系统的核心原则是，尽可能地让数据呆在原地、保持不动，同时尽可能地把承载计算任务的代码分发到离数据最近的地方（Executors 或计算节点），从而最大限

度地降低分布式系统中的网络开销。

每日一练

1. DAGScheduler 在创建 Tasks 的过程中，是如何设置每一个任务的本地性级别？
2. 在计算与存储分离的云计算环境中，Node local 本地性级别成立吗？你认为哪些情况下成立？哪些情况下不成立？

期待在留言区看到你的思考和答案，如果你的朋友也正急需搞清楚调度系统的工作原理，也欢迎你把这一讲转发给他，我们下一讲见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | DAG与流水线：到底啥叫“内存计算”？

下一篇 06 | 存储系统：空间换时间，还是时间换空间？

精选留言 (17)

写留言

L3nvy
2021-03-24

1.
位置信息通过特定的字符串前缀格式标识
executor_[hostname]_[executorid]
[hostname]
hdfs_cache_[hostname]...
展开

作者回复: 第一题给满分💯，看答案就知道认真去读源码了，赞一个～ 第二题也答对了。第二题，再想想，还有其他cases吗?

2 3

小学生敬亭山
2021-03-26

老师正例这个，先建map，再broadcast map 是不是一样的逻辑

作者回复: 好思路，完全没问题～

2

白音
2021-03-24

示例中关于读文件没太理解想请教下老师。
Source.fromFile 用于读本地文件，所以用spark读文件不是应该用 sc.textFile 来从hdfs目录读取? 或者示例的意思是在跑这段代码之前已经将模板文件分发到了集群每个executor本地吗?
展开

作者回复: 对，反例的情况，需要把模板文件分发到每个executors，每个executors都需要：1 读文件，建字典；2 在千亿样本上查找字典。

正例里面，driver读文件、建字典；executors上面，只做第二步。

3 2

2021-03-30

如果是我来写Label Encoding的话，在模板数据量不大的情况下，我会第一时间把模板数据转成Map之后广播出去。
看了老师的正例，感觉其实和广播的意思是差不多的吧

作者回复: 没错，和广播殊途同归～ 这里咱们为了强调调度系统，所以采用了这种实现方式哈～

1

尔雅
2021-03-25

吴老师,
偏函数是将多个操作转换为一个操作了吗,
如果不使用偏函数,就会造成dag 将操作拆分成多个 stages
造成调度的负载升高,可以这样理解吗

展开

作者回复: 不不，和偏函数没什么关系哈。我只是用Partial functions来举例，目的是为了讲调度系统。Partial functions不是关键哈，任务调度过程中，分发了哪些东西到executors，这个才重要哈，这些分发到东西，决定了executors的负载。

Partial functions只是方便举例，我顺手就拿过来用了，它只是一种实现形式哈～

dag拆分stages的关键因素是shuffle哈，shuffle是边界，是关键。

4 1

Geek_d794f8
2021-03-25

老师，数据尽量不动，比如有部分数据在节点A，那么移动计算难道不是要在节点A上启动Excutor才可以进行计算吗？但是Excutor不是在申请资源的时候就确定了在哪几个节点上启动Excutor吗？老师请指教一下

作者回复: 好问题，是这样的，资源调度和任务调度是分开的。

资源调度主要看哪些节点可以启动executors，是否能满足executors所需的cpu数量要求，这个时候，不会考虑任务、数据本地性这些因素。

资源调度完成之后，在任务调度阶段，spark负责计算每个任务的本地性，效果就是task明确知道自己应该调度到哪个节点，甚至是哪个executors。最后scheduler Backend会把task代码，分发到目标节点的目标executors，完成任务调度，实现数据不动代码动。

所以，二者是独立的，不能混为一谈哈～

1

Z宇锤锤
2021-04-01

Task的本地属性是由所属RDD的stage决定的。

展开

作者回复: 本地性是task粒度的哈，同一stage内部，不同task的本地性可能是不同的。再想想～

fsc2016
2021-03-31

老师，请教一个问题，在Graph Embedding中，基于用户行为的构造的物品概率转移矩阵，进行随机游走构造用户行为序列中，这个物品概率转移矩阵，如果物品维度很大，那么在driver和分发到executer端很容易oom，spark怎么解决类似这样问题的，想听听老师的想法!

展开

作者回复: 挺有意思的场景，其实这里没调度系统什么事，主要是数据倾斜的问题。这里的关键是构造by用户的物品转移序列样本。有了样本其实就可以拿来训练了。但是这里有data skew的风险，因为不同用户购买行为不一样，有的序列短，有的序列就会特别长，因此会skew。

解决思路分类讨论吧，分加盐还是不加盐。

不加盐，用window over user by 时间排序就好了，为了应对倾斜，要么把时间窗口缩短；要么把倾斜的user但拿出来处理，分而治之。

加盐，这个比较复杂，就是在user上加随机数，先shuffle把数据打散，然后各自构建转移序列，但最后还要merge回来、去盐化。merge的时候可以用join，然后把各自排序的时间序，用sort merge重新merge成一个序列（by user）。当然这里你可能需要定义udf来专门干merge这个事，除了物品还要带上时间，因为你要按照时间顺序去merge。虽然复杂，但不失为一种办法。

哪种方法最好，还有看你的数据分布，欢迎继续讨论，另外有了结果，不妨share出来哈～

2

小学生敬亭山
2021-03-26

老师好，为什么第二个示例（也就是正例），建hashmap这个代码就一定是在driver端执行，然后代码再进入调度系统我不是太明白?
什么时候代码会在driver端执行?

作者回复: 这一步比较重，从数据处理的最终目的来说，这一步算是开销，放在driver端算一次是最佳策略，避免这个开销在executors断反复计算。

没有被apply到分布式数据集的计算，都是在driver执行。比如常见的数组、list、字符串，等等。rdd. dataframe. 这些分布式数据集上apply的计算逻辑，才会分发到executors去执行。

Geek_f65511
2021-03-26

收获很大，以前这些地方理解的不细，现在一下都清晰了。

牛红灯
2021-03-25

老师,只有触发action算子的才在driver上运行吧,第二种方法中也没有提到是action呀,为什么读文件是在driver上呢

展开

作者回复: 是这样，模板文件是本地文件，因此可以从driver读进内存、构建字典，最后从driver分发到executors。咱们这里为了讲调度系统、强调任务分发，所以并没有给出完整的代码。所以你看到dataset那里apply函数就结束了，后面也没有写是把数据落盘、还是collect这些actions，目的还是让大家把注意力集中到与调度开销息息相关的那几行代码～

2

来世愿做友人 A

2021-03-24

第一题：因为是为每个 partition 建立一个task，所以在建立task之前，都会获取每个partition的位置偏好信息。首先判断 rdd 是否被缓存过，通过 rddId + splitIndex 组合成 blockId 判断。如果没有，判断preferredLocations，看起来是判断是否 checkpoint 过。如果还没有，向上获取父rdd，如果是窄依赖，循环上面的判断逻辑。这里想问个问题，代码里直到task分发，似乎没有看到关于shuffle的位置偏好。比如中间有个shuffle过程，...

展开

作者回复: 如果你说的是shuffle read阶段的locality，咱们换个角度思考这个问题。shuffle map阶段，每个map task把中间文件写到本地盘。shuffle read阶段，每个reduce task需要从集群的所有节点拉数据，走网络。这个过程是由shuffle的实现机制决定的。

因此，从reduce视角看过去，所有中间文件都是它的“数据源”，这些数据源，散落在集群的每一个节点，因此，每个reduce task的locality，在最好的情况下，能做到rack local，最差的时候，那就是any。

正是因为这样，所以我们会强调，竭尽全力避免shuffle。因为它的实现机制，决定了reduce task的计算一定会让数据走网络。

不知道这么说能不能回答你的问题哈～ 要是我理解错了，就再at我，咱们继续讨论哈～

1

October

2021-03-24

老师，您提到如果taskScheduler采用Fair调度策略对不同stages进行调度，可以为不同的用户配置不同的调度池，刚开始这个地方有些不理解，同一个应用程序中，怎么会有不同的用户？ 于是查了一下官网，官网貌似说的是这里的用户不是程序提交的那个用户，这里的用户对应提交job的一个线程，不知道自己理解的是否正确。 另外，如果我的理解正确的话，在同一个应用程序中使用不同的线程提交job，这个使用方式，我目前还没有见过...

展开

作者回复: 非常好的问题！非常好！通过问题就能看出来，这一篇读的非常认真，赞一个先～ 是这样，你查的没错，首先用户的说法确实容易造成困惑，其实就是应用中多个不同的job，换句话说，就是你应用中的多个actions。开发的时候，你可以把job assign到某个调度池，从而区分不同job在资源争抢上的优先级。

回答你最后的问题，换一种问法，其实就是什么场景下，一个应用会有多个job。其实这种cases

很多，我最先想到的，就是etl，因为etl往往有多个目的，读取多个数据源，输出多种不同阶段的数据，这是常有的事，因此etl至少是一个。

另外一个，机器学习中的样本过程，通过一个应用生产多个schema的训练样本，这也会涉及到多个job。

你可以沿着这个思路，再想想还有没有其他cases哈～

October
2021-03-24

老师，taskscheduler的调度是把taskset根据一定的调度算法分配给不同的executor，那么我有一个问题，executor资源的申请是在任务的三级调度之前就已经申请好了吗？如果是这样的话，executor的申请时，有没有考虑优先在数据所在的节点开启executor?

展开

作者回复: 好问题，有一点需要特别注意，spark的资源调度和任务调度是分开的哈。也就是说，不论你用standalone、yarn还是mesos，spark在申请硬件资源的时候，以cpu、memory量化申请executors的过程，是先于任务调度的。用你的话说，executors已经提前申请好了，申请executors的时候，只看cpu和memory是否满足要求，不会考虑locality这些与任务有关的细节。咱们这一讲，focus在任务调度的讲解，其实在任务调度之前，资源调度就已经完成了。

October
2021-03-24

两种实现方式都能理解，但是现阶段很难写出像老师这样的非常scala风格的代码。 如果是我的话，实现方式可能会定义一个类，这个类包含这个字典属性，在driver端都文件，完成字典属性的初始化，在exexutor查字典。

展开

作者回复: 语言不重要哈，只要能get到调度系统的工作原理、调度开销，能理解不同实现方式在运行时的计算过程、识别分布式执行效率最高的实现方式，就达到目的了哈～ 至于说是oo、还是fp，不重要。黑猫白猫，能抓耗子就是好猫～

对方正在输入。。。

2021-03-24

老师，对于第二点，如果计算和存储分离的情况，我理解的是：对于shuffleMapStage来说，node local不成立；对于resultStage来说，从外部数据源读或者从shuffle数据读，node local也都不成立

作者回复: 对，物理分离的话，都不成立～

毓秀靓泽

2021-03-24

收获颇大，以前没往这块儿想

展开